CS EE World
https://cseeworld.wixsite.com/home
28/34 (A)
May 2025

Submitter info:
My main tip is to find an exemplar that scored well and has a similar topic, and align your own EE with their structure and methodology.
If you have any questions, feel free to reach out at ponpouiwui [at] gmail [dot] com.

Computer Science Extended Essay:

Investigating the Effect of Spatial Dimensions on Time Complexity and Search Effort of Shortest Path Algorithms

---

Research Question: How does incorporating additional spatial dimensions impact the time complexity and search effort of A* and Breadth-First Search (BFS) algorithms in grid-based environments?

Candidate Code: lrp752

Session: May 2025

Word Count: 3989

Table of Contents

1. Introduction **3**

2. Background Information **4**

2.1 Time Complexity 4

2.2 Breadth-First Search 6

2.3 A* Algorithm 12

3. Experiment **19**

3.1 Hypothesis 19

3.2 Applied Theory 20

3.3 Variables 21

3.4 Procedure 23

4. Experimental Results **24**

4.1 Data Collection Averages 24

4.2 Data Visualization and Analysis 25

5. Evaluation **31**

5.1 Experiment’s Implications for Further Dimensions 33

5.2 Limitations 33

6. Conclusion **35**

7. Bibliography **37**

8. Appendix **39**

1. Introduction

The process of finding the shortest path from point A to point B is an essential part of daily life. In the age of technology, this process, known as pathfinding, plays a critical role in fields such as transportation, robotics, and video games. Effective pathfinding influences safety in autonomous vehicles, cost efficiency in delivery drones, and immersion in video games. This process relies on algorithms to find the optimal route based on criteria such as shortest distance, lowest cost, or safest path. The choice of algorithm depends on the specific application (Yap 2022). For example, a drone delivering packages may prioritize minimizing travel costs, while a video game non-playable character may prioritize faster computation, even at the expense of optimality.

Pathfinding algorithms like A* and Breadth-First Search (BFS) are widely used across these fields. However, many traditional algorithms assume movement is restricted to two dimensions, such as flat road networks. This assumption breaks down in 3D environments, such as drone navigation or complex games, where vertical movement introduces new challenges. This essay aims to explore the challenges of pathfinding in grid-based environments by investigating how A* and BFS algorithms perform as the number of spatial dimensions increases. The empirical evaluation will focus on 2D and 3D environments, while further dimensions, including 4D and beyond, will be analyzed theoretically. This will provide insights into how the search space expands and the algorithms' computational complexity scales with increasing dimensions.

2. Background Information

Pathfinding algorithms traverse through nodes (vertices) across edges (paths). Edges have certain weights that determine their cost for traversing from one vertex to another. Pathfinding algorithms aim to minimize the total cost of traveling from a start node to an end node. These scenarios can be represented in various ways, such as weighted graphs and grid structures. Grid-based pathfinding is an effective form of visualization, as it simplifies the concept of movement. In this scenario, nodes are represented by squares. In two dimensions, nodes are connected with four adjacent vertices (unless located next to boundaries), representing edges. In higher-dimensional spaces, such as 4D, nodes are no longer limited to connections in two nor three directions, (x, y) or (x, y, z). A node in a 4D grid has neighbors along an additional axis, *w*, representing movement in the fourth dimension. A node can connect to up to 8 neighbors in a 4D grid, as it has positive and negative connections for each of the four spatial axes. With each added dimension, the number of neighbors and overall search complexity grows exponentially.

2.1 Time Complexity

Big O notation describes how an algorithm's runtime or space grows with input size, providing an upper bound on its time complexity (Huang 2020). It reflects worst-case performance, helping assess scalability, which is the algorithm's ability to handle increasing grid size and dimensionality.

We can write that a positive function $f(N) = O(g(N))$ if there is a constant C such that for large values of N, the following holds:

$f(N) \leq C * g(N)$ (DataCamp 2024)

Following this definition, the runtime complexity of BFS and A* can be expressed in terms of Big O.

Algorithm	**Runtime Complexity (2D)**	**Explanation**
Breadth-First Search	$O(V + E)$ where *V* is the number of nodes and *E* is the number of edges in the grid (Tong and Tang 2023).	BFS visits each node exactly once, which results in a time complexity of $O(V)$. For each node, BFS examines all of its adjacent edges to explore neighboring nodes. This results in an additional time complexity of $O(E)$. Since BFS processes each node and its corresponding edges, the overall time complexity is $O(V + E)$, which accounts for both the nodes and the edges in the graph.
A*	$O((V + E)(log_2 V))$ where *V* is the number of nodes and *E* is the number of edges in the grid. This complexity arises due to the algorithm's use of a priority queue to manage nodes during exploration.	A* explores nodes by expanding the node with the lowest estimated cost (using heuristic estimation goal), contributing to complexity of $O(V)$. For each node, A* examines its neighboring edges to explore possible paths. This examination of edges contributes to the $O(E)$ complexity. The use of a priority queue to manage the open list of nodes introduces a logarithmic factor ($log_2 V$) to the complexity. Each insertion or extraction operation on the priority queue takes $O(log_2 V)$ time.

Table 1: BFS and A* Time Complexity Comparison

In higher dimensions, both BFS and A* face increased runtime due to exponential growth in the search space. Each node connects to more neighbors—4 in 2D, 6 in 3D, and 8 in 4D—further expanding the search space as dimensionality increases. This contributes directly to the overall time complexity scaling with V, the number of nodes.

2.2 Breadth-First Search

BFS is an uninformed search algorithm that expands outward from a starting node, commonly used in network broadcasting and graph traversal. It does not require knowledge of the goal, making it versatile but inefficient at prioritizing promising paths (GeeksforGeeks 2024). In higher dimensions, BFS must account for additional neighbors along each axis. For example, in 4D, nodes have 8 neighbors instead of 4 in 2D, which expands the search space and increases search effort.

```
1  BFS(grid, startRow, startCol, endRow, endCol):
2      queue = empty queue // Create a queue to explore nodes
3      visited = empty set // Create a set to track visited nodes
4      parent = empty map // Create a map to track each node's parent
5
6      queue.enqueue((startRow, startCol)) // Add the start node to the queue
7      visited.add((startRow, startCol)) // Mark the start node as visited
8      parent[(startRow, startCol)] = None // The start node has no parent
9
10     while queue is not empty: // While there are nodes to explore
11         currentNode = queue.dequeue() // Get the node at the front of the queue
12         currentRow, currentCol = currentNode // Get the row and column of the current node
13
14         if currentRow == endRow and currentCol == endCol: // Check if we reached the end
15             return reconstructPath(parent, (endRow, endCol)) // Return the path
16
17         // Check each neighboring node (up, down, left, right)
18         for direction in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
19             newRow = currentRow + direction[0]
20             newCol = currentCol + direction[1]
21
22             if isValid(grid, newRow, newCol) and (newRow, newCol) not in visited:
23                 visited.add((newRow, newCol)) // Mark the node as visited
24                 queue.enqueue((newRow, newCol)) // Add the node to the queue
25                 parent[(newRow, newCol)] = (currentRow, currentCol) // Record the parent
26
27     return "No path found" // If no path is found
28
29 // Helper function to check if a node is within bounds and walkable
30 isValid(grid, row, col):
31     return row >= 0 and row < grid.height and col >= 0 and col < grid.width and grid[row][col] is walkable
32
33 // Helper function to reconstruct the path from end to start
34 reconstructPath(parent, endNode):
35     path = empty list
36     while endNode is not None:
37         path.prepend(endNode) // Add the node to the path
38         endNode = parent[endNode] // Move to the parent
39     return path // Return the path
```

Figure 1: Pseudocode for 2D grid-based Breadth-First Search (ChatGPT-4o 2024)

The pseudocode in Figure 1 outlines the BFS algorithm for a 2D grid. It initializes an empty queue (line 2), a set to track visited nodes (line 3), and a map to store parent nodes (line 4). The start node is added, marked as visited, and assigned no parent (lines 6–8).

The main loop (line 10) runs until the queue is empty, dequeuing a node (line 11) and checking if it is the goal (lines 13–15). If so, reconstructPath() (lines 34–39) traces back through the parent map. Otherwise, BFS explores the four orthogonal neighbors (lines 17–18), checking their validity (lines 30–32), and marking valid nodes as visited, enqueuing them, and assigning parents (lines 22–24). If no path is found when the queue empties, BFS returns failure (line 26).

BFS guarantees the shortest path in unweighted grids by expanding nodes uniformly layer by layer, systematically covering all possible paths in order of distance from the start.

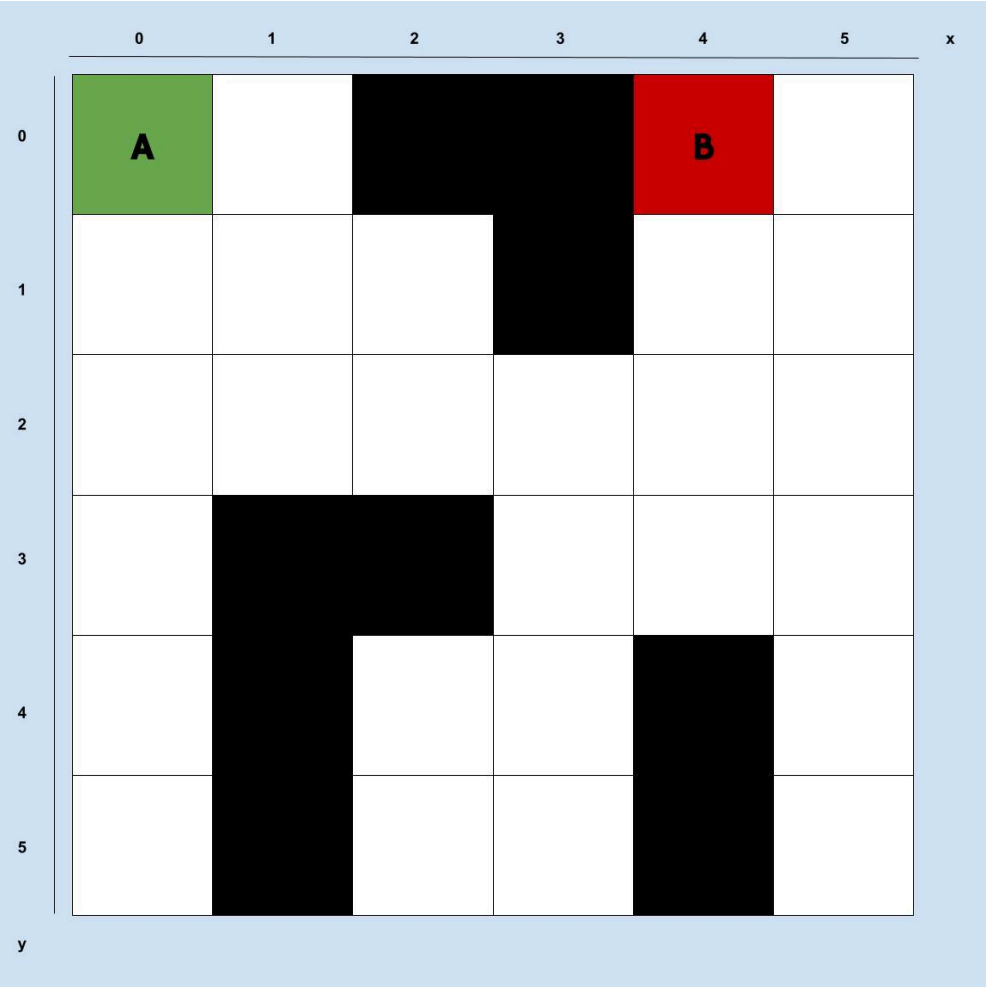


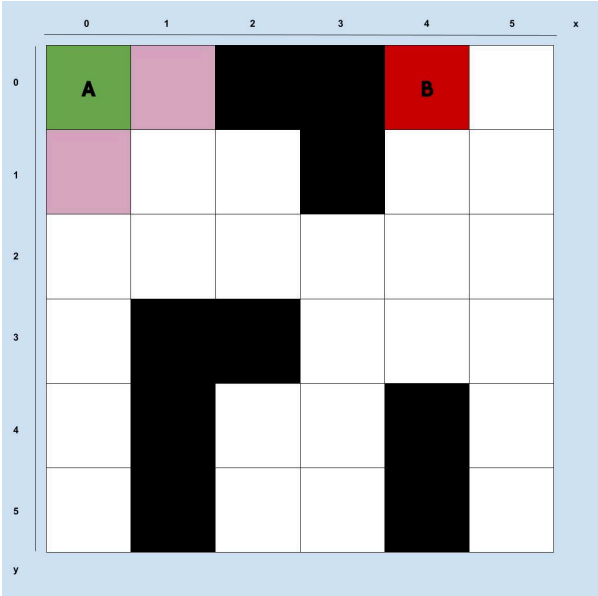
Figure 2: Unweighted 2 dimensional maze grid

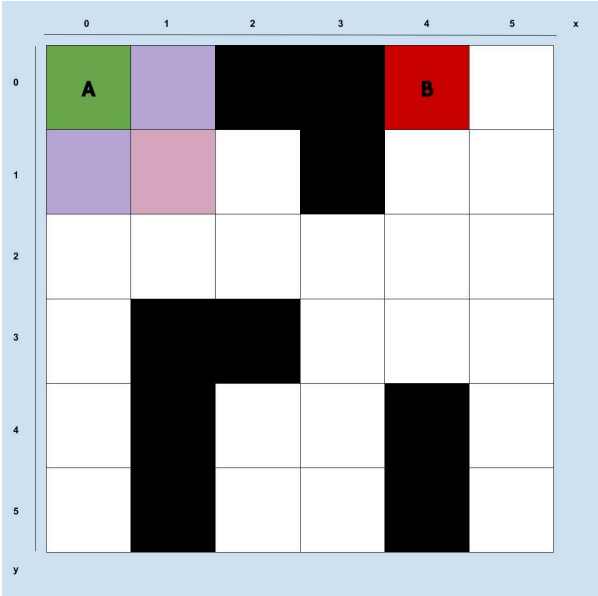
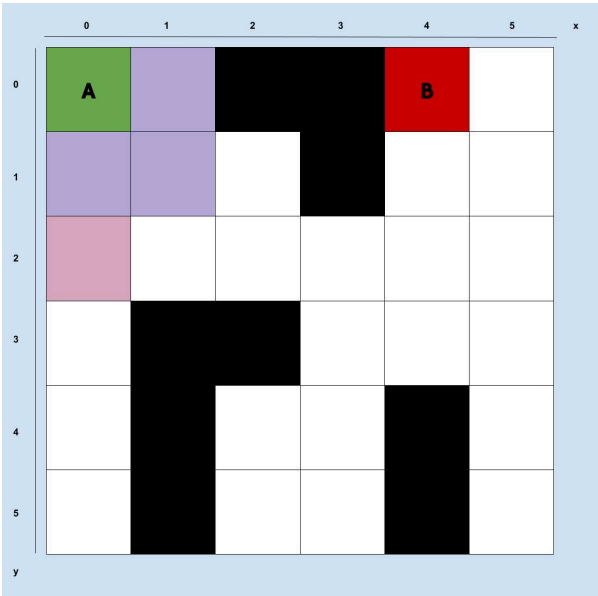
Mazes are often used as standardized test environments for pathfinding algorithms because they offer a controlled yet challenging scenario that effectively highlights the strengths and weaknesses of different algorithms. The maze in Figure 2 lies on an x-y plane.

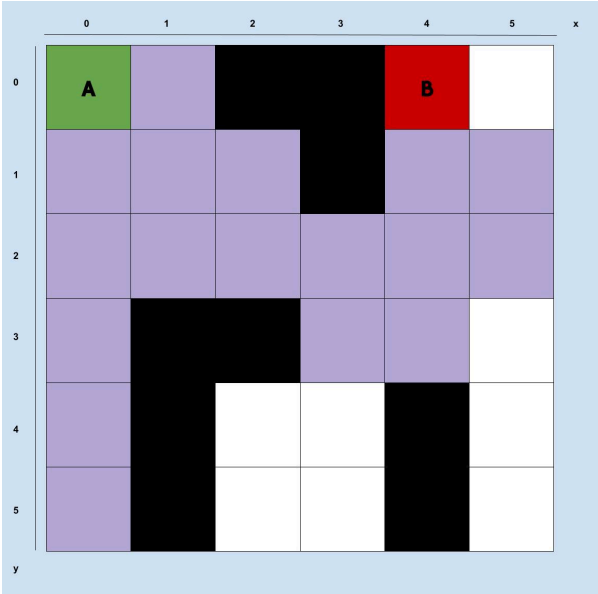
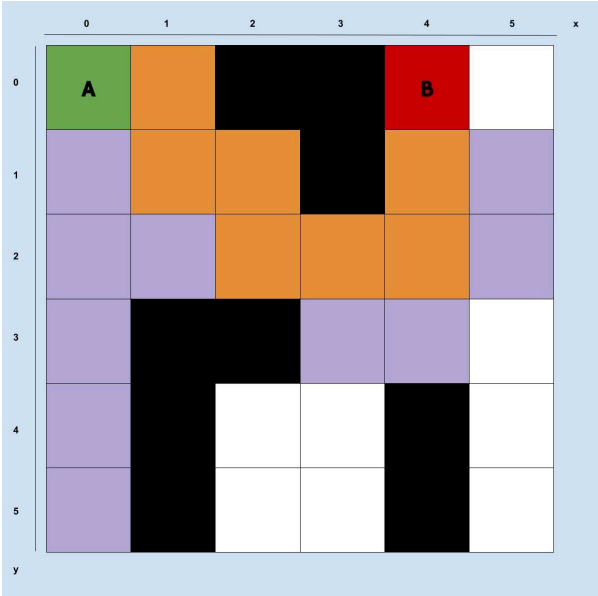
Below is an example of how BFS iterates through the maze to reach the end node (B) in Figure 2.

Key
Green node (A): Start node (0, 0). Red node (B): End node (4, 0). Black node: Obstacle (invalid node) Pink node: Newly added node to visited set. Purple node: Node previously stored in visited set. Orange node: Reconstructed path node.

Figure 3: Key for visual examples of pathfinding algorithms in 2 dimensions.

Graph	**Explanation**
	The start node is initialized as (0, 0), the visited set is initialized empty, and the queue only stores the start node (A). **Iteration 1:** In the first iteration, the current node is initialized as (0, 0), which is node A. It then explores the adjacent nodes, (1, 0) and (0, 1) (visualized with purple boxes), confirming they are valid, marking them as visited, and adding them to the queue. Visited Set: {(0, 0), (1, 0), (0, 1)} Queue: [(1, 0), (0, 1)] Parent Set: {(0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0)} The parent set at this stage reflects the first two

	valid neighbors, (1, 0) and (0, 1), which were added to the queue. Both nodes have (0, 0) as their parent since they were discovered directly from the start node. The parent set will be later used to reconstruct the path from the start node to the end node.
	**Iteration 2:** Current Node: (1, 0) Neighbor Nodes: [(2, 0), (0, 0), (1, 1)] (newRow, newCol pairs). Node (0, 0) has already been explored, so it is not re-added to the queue. Node (0, 2) is an invalid node (obstacle), so it is avoided. Newly visited nodes are visualized with pink. Visited set: {(0, 0), (1, 0), (0, 1), (1, 1)} Queue: [(0, 1), (1, 1)] Parent Set: {(0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0)}
	**Iteration 3:** Current Node: (0, 1) Neighbor Nodes: [(2, 0), (0, 0), (1, 1)] (newRow, newCol pairs). (0, 0) and (1, 1) already visited. Visited set: {(0, 0), (1, 0), (0, 1), (1, 1), (0, 2)} Queue: [(1, 1), (0, 2)] Parent Set: {(0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0), (0, 2): (0, 1)}

	This process continues until the end node is visited. The corresponding graph illustrates the visited nodes by the BFS algorithm. In total, 19 nodes are visited.
	The algorithm reconstructs the path (Orange tiles) using the parent set. The path found by BFS is 8 nodes long.

This example demonstrates BFS's systematic layer-by-layer exploration, ensuring the shortest path in an unweighted grid by visiting all equidistant nodes before proceeding. However, this exhaustive approach results in higher computational effort, requiring 19 node visits to find an 8-tile path. While this guarantees optimality, BFS's performance can degrade significantly in larger grids due to exponential growth in visited nodes.

2.3 A* Algorithm

A* is a widely used pathfinding algorithm known for efficiently finding the shortest path between two points in a grid. Unlike BFS, it is informed, using knowledge of the end node. It combines Dijkstra's approach of exploring nodes by shortest accumulated distance with a heuristic estimating the cost to the goal, allowing it to prioritize promising paths (Östberg 2021). This balance makes A* both optimal, always finding the shortest path if one exists, and complete.

Heuristics

The heuristic function is a core component of A*, allowing it to prioritize promising paths over less efficient ones. Unlike uninformed algorithms like BFS, A* uses a heuristic to estimate the cost from each node to the goal, reducing the number of nodes expanded and overall computational effort (Johnson 1997–2024). The heuristic, $h(n)$, estimates the minimum cost from node *n* to the goal, which is combined with $g(n)$, the actual cost from the start, to compute the total estimated cost:

$$f(n) = g(n) + h(n).$$

Nodes with lower $f(n)$ values are prioritized for exploration, as they are considered more likely to lead to the goal with minimal additional cost.

In this essay, I will be using the Manhattan distance heuristic, which is particularly well-suited for grid-based environments where movement is restricted to horizontal and vertical directions. The Manhattan distance provides a simple and admissible approximation of the path cost and in 3 dimensions is defined as:

$$h(n) = |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|,$$

where (x_1, y_1, z_1) represents the coordinates of the current node *n*, and (x_2, y_2, z_2) represents the coordinates of the goal node.

The Manhattan distance heuristic suits grid-based pathfinding because it reflects movement restricted to horizontal and vertical steps, aligning with the grid's structure. In contrast, options such as Euclidean distance assume diagonal travel, making it less accurate for axis-aligned movement (Johnson 1997–2024). This alignment between the heuristic and the grid's movement rules helps improve the efficiency and accuracy of the A* algorithm.

The heuristic function is critical in higher-dimensional environments, as it helps mitigate the increased computational burden by prioritizing paths that are likely to lead to the goal efficiently.

A* Explanation

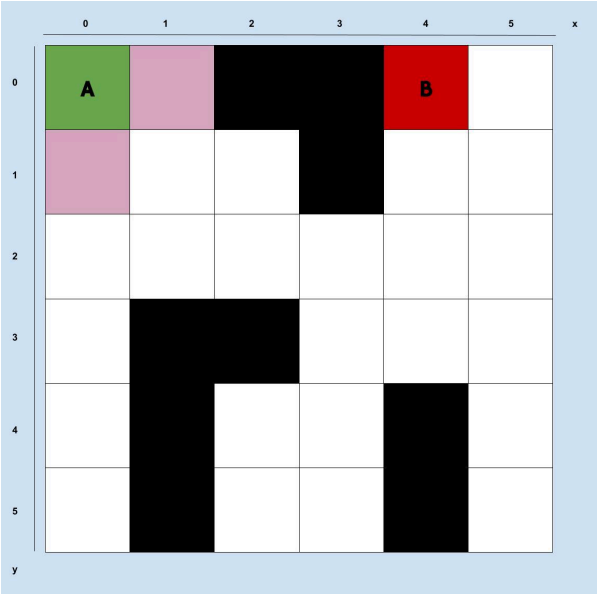
```
1  AStar(grid, startRow, startCol, endRow, endCol):
2      openSet = empty priority queue  // Nodes to be explored, prioritized by cost
3      openSet.enqueue((startRow, startCol), 0)  // Add the start node with priority 0
4
5      cameFrom = empty map  // To reconstruct the path
6      gScore = map with default value ∞  // Cost from start to each node
7      gScore[(startRow, startCol)] = 0  // Cost from start to start is 0
8
9      fScore = map with default value ∞  // Estimated total cost from start to goal
10     fScore[(startRow, startCol)] = heuristic(startRow, startCol, endRow, endCol)
11
12     while openSet is not empty:  // While there are nodes to explore
13         current = openSet.dequeue()  // Node in openSet with the lowest fScore
14         currentRow, currentCol = current
15
16         if currentRow == endRow and currentCol == endCol:  // Check if we reached the end
17             return reconstructPath(cameFrom, (endRow, endCol))  // Return the path
18
19         for direction in [(0, 1), (0, -1), (1, 0), (-1, 0)]:  // Explore neighbors (right, left, down, up)
20             neighborRow = currentRow + direction[0]
21             neighborCol = currentCol + direction[1]
22
23             if isValid(grid, neighborRow, neighborCol):  // Check if the neighbor is within bounds and walkable
24                 tentative_gScore = gScore[(currentRow, currentCol)] + distance(currentRow, currentCol, neighborRow, neighborCol)
25
26                 if tentative_gScore < gScore[(neighborRow, neighborCol)]:  // If we found a shorter path to the neighbor
27                     cameFrom[(neighborRow, neighborCol)] = (currentRow, currentCol)  // Record the best path to the neighbor
28                     gScore[(neighborRow, neighborCol)] = tentative_gScore
29                     fScore[(neighborRow, neighborCol)] = tentative_gScore + heuristic(neighborRow, neighborCol, endRow, endCol)
30
31                     if (neighborRow, neighborCol) not in openSet:  // Add neighbor to openSet if not already present
32                         openSet.enqueue((neighborRow, neighborCol), fScore[(neighborRow, neighborCol)])
33
34     return "No path found"  // If openSet is empty and no path was found
35
36 // Helper function to check if a node is within bounds and walkable
37 isValid(grid, row, col):
38     return row >= 0 and row < grid.height and col >= 0 and col < grid.width and grid[row][col] is walkable
39
40 // Helper function to reconstruct the path from end to start
41 reconstructPath(cameFrom, current):
42     path = empty list
43     while current is not None:
44         path.add(current)
45         current = cameFrom[current]
46     return reverse(path)
47
48 // Heuristic function to estimate distance from a node to the end
49 heuristic(row, col, endRow, endCol):
50     return abs(row - endRow) + abs(col - endCol)  // Manhattan distance
51
52 // Function to calculate the distance between two nodes
53 distance(row1, col1, row2, col2):
54     return 1  // Assuming uniform cost between adjacent nodes
```

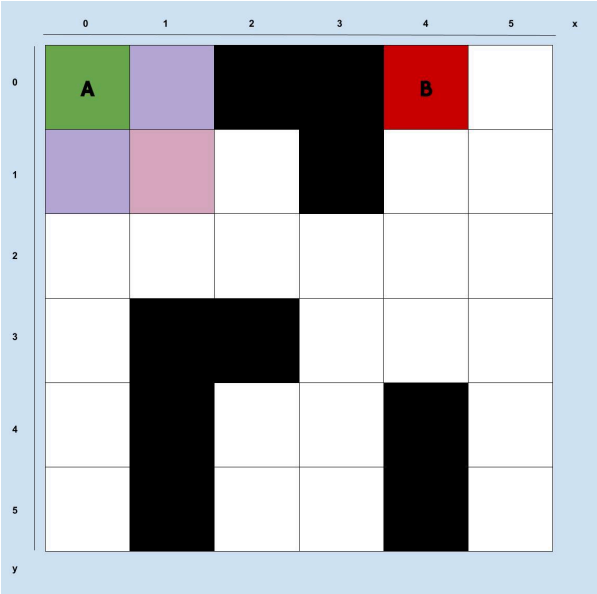
Figure 4: Pseudocode for 2D grid-based A* algorithm (ChatGPT-4o, 2024)

The pseudocode in Figure 4 outlines the A* algorithm for a 2D grid. It begins by initializing a priority queue to explore nodes by cost (line 2) and maps to store movement cost (gScore) and estimated total cost (fScore) (lines 6–9). The main loop (line 11) processes nodes until the queue is empty. Each iteration dequeues the node with the lowest fScore (line 12) and either reconstructs the path if the goal is reached (lines 14–16) or expands neighboring nodes (lines

18–19). Valid neighbors (lines 38–40) have their movement cost updated (line 24) and are added to the queue if a shorter path is found (lines 26–30). The heuristic function (lines 47–49) estimates the remaining cost using Manhattan distance. If no path is found, the algorithm terminates (line 32). This structure allows A* to prioritize promising paths, aiming for efficient pathfinding in grid-based environments.

Below is an example of how BFS iterates through the maze to reach the end node (B) (Refer to Figure 2 and 3):

Graph	**Explanation**
	The start node (0, 0) is added to the priority queue with fScore = 0. **Iteration 1:** In the first iteration, the current node is initialized as (0, 0), which is node A. Valid neighbor nodes: ● (1, 0): ○ gScore = current node gScore + distance from current to neighbor node. $gScore = 0 + \|1 - 0\| + \|0 - 0\| = 1$ ○ fScore = gScore + $h(n)$ (distance to end node). Thus, $fScore = 1 + \|1 - 4\| + \|0 - 0\| = 4$ ● (0, 1): ○ $gScore = 0 + \|0 - 0\| + \|1 - 0\| = 1$ ○ $fScore = 1 + \|0 - 4\| + \|1 - 0\| = 6$ Visited Set: {(0, 0), (1, 0), (0, 1)} Priority Queue: [(1, 0): 4, (0, 1): 6] Parent Set: {(0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0)}



Iteration 2:
Current Node: (1, 0)

Valid neighbor node:

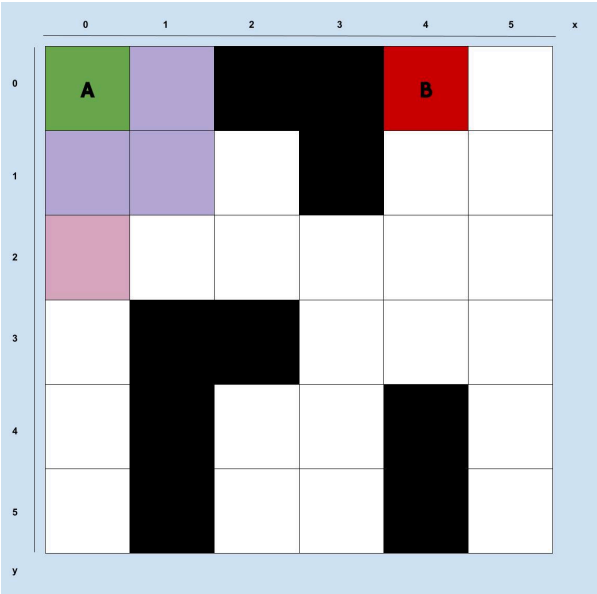
- (1, 1):
  - gScore = $1 + |0 - 0| + |0 - 1| = 2$
  - fScore = $2 + |1 - 4| + |1 - 0| = 6$

Node (0, 0) has already been explored and node (0, 2) is an invalid node (obstacle), so they are omitted. Newly visited nodes are visualized with pink.

Visited set: {(0, 0), (1, 0), (0, 1), (1, 1)}

Priority Queue: [(0, 1): 6, (1, 1): 6]

Parent Set: {(0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0)}



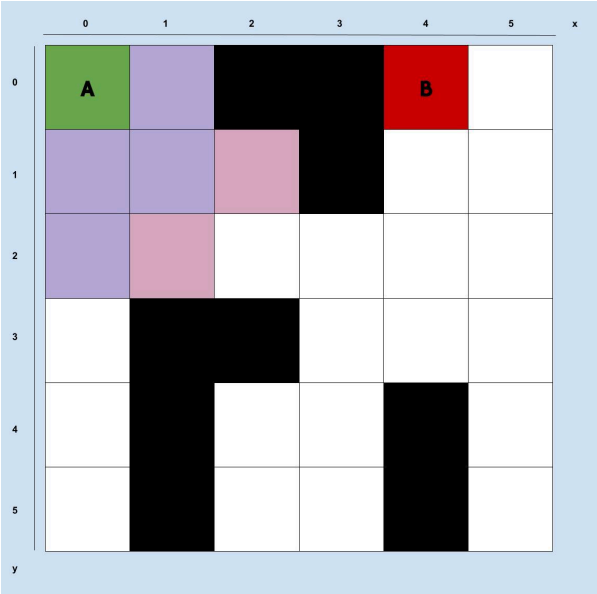
Iteration 3:
Current Node: (0, 1)
Valid neighbor node:

- (0, 2):
  - gScore = $1 + |0 - 0| + |1 - 2| = 2$
  - fScore = $2 + |0 - 4| + |2 - 0| = 8$
- (1, 1):
  - gScore = $1 + |0 - 1| + |1 - 1| = 2$
  - fScore = $2 + |1 - 4| + |1 - 0| = 6$

Visited set: {(0, 0), (1, 0), (0, 1), (1, 1), (0, 2)}

Priority Queue: [(1, 1): 6, (0, 2): 8]

Parent Set: {(0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0), (0, 2): (0, 1)}



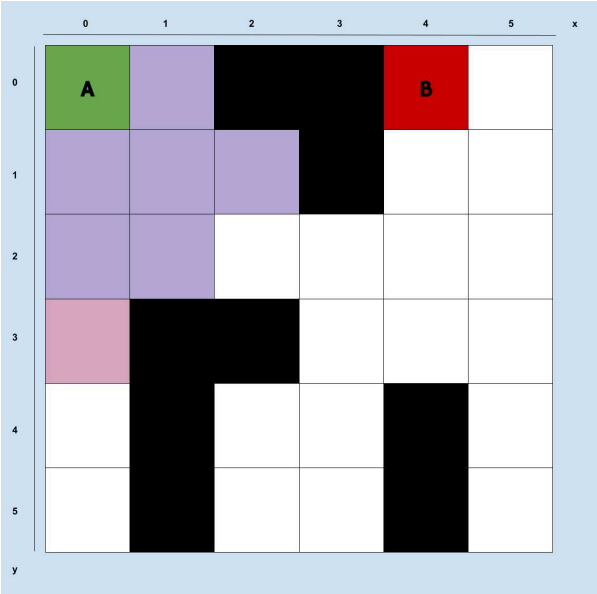
Iteration 4:
Current Node: (1, 1)
Valid neighbor nodes:

- (2, 1):
  - gScore = $2 + |1 - 2| + |1 - 1| = 3$
  - fScore = $3 + |2 - 4| + |1 - 0| = 6$
- (1, 2):
  - gScore = $2 + |1 - 1| + |1 - 2| = 3$
  - fScore = $3 + |1 - 4| + |2 - 0| = 8$

Visited set: {(0, 0), (1, 0), (0, 1), (1, 1), (0, 2), (2, 1), (1, 2)}

Priority Queue: [(2, 1): 6, (0, 2): 8, (1, 2): 8]

Parent Set: {(0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0), (0, 2): (0, 1), (2, 1): (1, 1), (1, 2): (1, 1)}



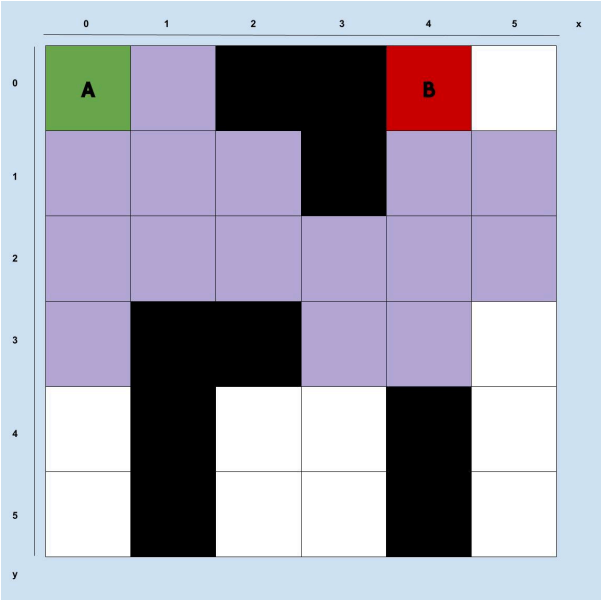
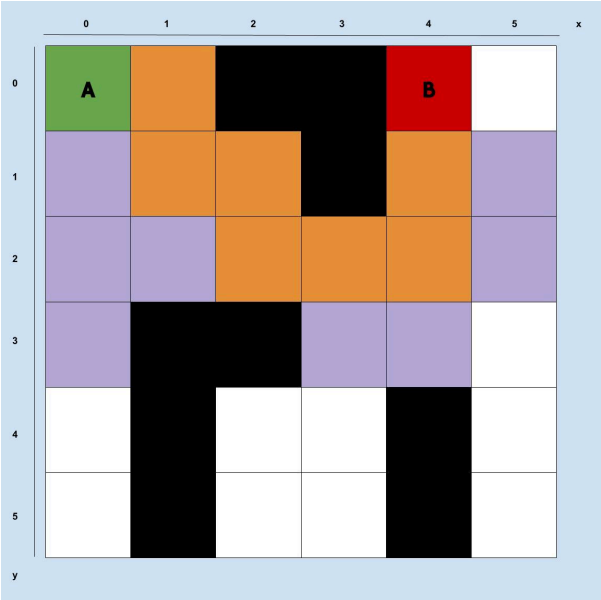
Iteration 4:
Current Node: (2, 0)
Valid neighbor nodes:

- (2, 1):
  - gScore = $2 + |2 - 2| + |0 - 1| = 3$
  - fScore = $3 + |2 - 4| + |1 - 0| = 6$
- (0, 3):
  - gScore = $2 + |2 - 3| + |0 - 0| = 3$
  - fScore = $3 + |0 - 4| + |3 - 0| = 10$

Visited set: {(0, 0), (1, 0), (0, 1), (1, 1), (0, 2), (2, 1), (1, 2)}

Priority Queue: [(2, 1): 6, (0, 2): 8, (1, 2): 8, (3, 0): 10]

Parent Set: {(0, 0): None, (1, 0): (0, 0), (0, 1): (0, 0), (1, 1): (1, 0), (0, 2): (0, 1), (2, 1): (1, 1), (1, 2): (1, 1)}

	This process continues until the end node is visited. The corresponding graph illustrates the visited nodes by the A* algorithm. In total, 16 nodes are visited.
	The algorithm reconstructs the path (Orange tiles) using the parent set. The path found by A* is 8 nodes long.

The results from the 2D examples show that both A* and BFS successfully identified the shortest path of 8 nodes. However, A* achieved this while visiting only 16 nodes compared to BFS's 19, demonstrating that A*'s heuristic reduces unnecessary exploration, while BFS's exhaustive search visits all reachable nodes. While this is well-documented in 2D, far less research explores

how these algorithms behave in higher dimensions such as 3D, where each node can have up to 6 neighbors instead of 4.

3. Experiment

This study investigates the performance of Breadth-First Search (BFS) and A* algorithms in grid-based pathfinding environments, specifically analyzing how the increase from two to three dimensions affects their time complexity and search effort. The experiment measures the total nodes in the grid (x), with the total nodes visited and the algorithm's execution time (y) to quantify their scalability across dimensions.

3.1 Hypothesis

I hypothesize that the execution time and search effort of both algorithms will increase significantly in three dimensions due to the exponential growth of the search space. BFS, which explores all reachable nodes uniformly, is expected to experience a sharper increase in nodes visited compared to A*, which uses its heuristic to guide the search. This difference will result in A* demonstrating greater scalability and efficiency in higher dimensions.

3.2 Applied Theory

The time complexity of BFS is $O(V + E)$. In two dimensions, each node connects to up to four neighbors ($E \approx 4V$), while in three dimensions, this increases to six neighbors ($E \approx 6V$). Consequently, the number of edges grows linearly with the dimensionality, contributing to a

larger $O(E)$ term. However, the exhaustive nature of BFS results in the algorithm traversing a significant number of unnecessary nodes, especially in obstacle-dense environments.

A*, with a complexity of $O((V + E)log_2 V)$, with the logarithmic factor arising from its use of a priority queue to select the next node. As dimensions increase, the branching factor, which is the average number of edges per node, grows from $b = 4$ in 2D to $b = 6$ in 3D (Cox 2024). This increases the search space, but A* limits this growth through heuristic pruning, balancing expanded search space with the cost of maintaining the priority queue.

3.3 Variables

Throughout the experiment, the size of the grids will be changed. This variable directly affects the performance of the algorithms due to the increase in the number of nodes and edges. The other variable that will be changing is the environment layout; specifically, the distribution of obstacles within each grid. Obstacles are generated randomly with a fixed density of 40% and represent inaccessible nodes. Each grid size will have five trials, and for each trial a new randomly generated grid layout will be used. By varying the obstacle layout, the experiment captures the algorithms' ability to adapt to different environments while ensuring the results are representative and not biased toward specific configurations.

The first dependent variable is execution time, which measures the total time taken by each algorithm to find the shortest path from the start node to the goal node. This metric is a key indicator of computational efficiency and reflects how well each algorithm handles increasing

complexity. The other dependent variable is the number of nodes visited during the search process. This variable provides insight into the search effectiveness.

The experiments were conducted using Visual Studio Code 1.96.2 on a MacBook Air M3 (10-core CPU, 8-core GPU, 16GB RAM) running macOS. These specifications were consistent throughout to maintain controlled conditions.

The start and end nodes are fixed for all trials, with the start node always located at the grid's origin (0, 0) for 2D and (0, 0, 0) for 3D, and the end node positioned at the farthest corner of the grid.

```
start_node = (0, 0, 0)
goal_node = (GRID_SIZE - 1, GRID_SIZE - 1, GRID_SIZE - 1)
```

Additionally, the obstacle density is kept constant at 40%, maintaining a consistent level of environmental complexity across all trials. Each grid size is tested across five trials to account for variability in obstacle placement, ensuring statistical reliability. Furthermore, the number of nodes for each grid size comparison between 2D and 3D will remain constant. For example, for a grid size of 64^2 in two dimensions, a grid size of 16^3 will be used, since $64^2 = 16^3 = 4096$. Lastly, the 5 grid sizes chosen are all exponentially increasing integer cubic and square roots, to ensure a controlled testing environment, and to highlight the effects of scaling.

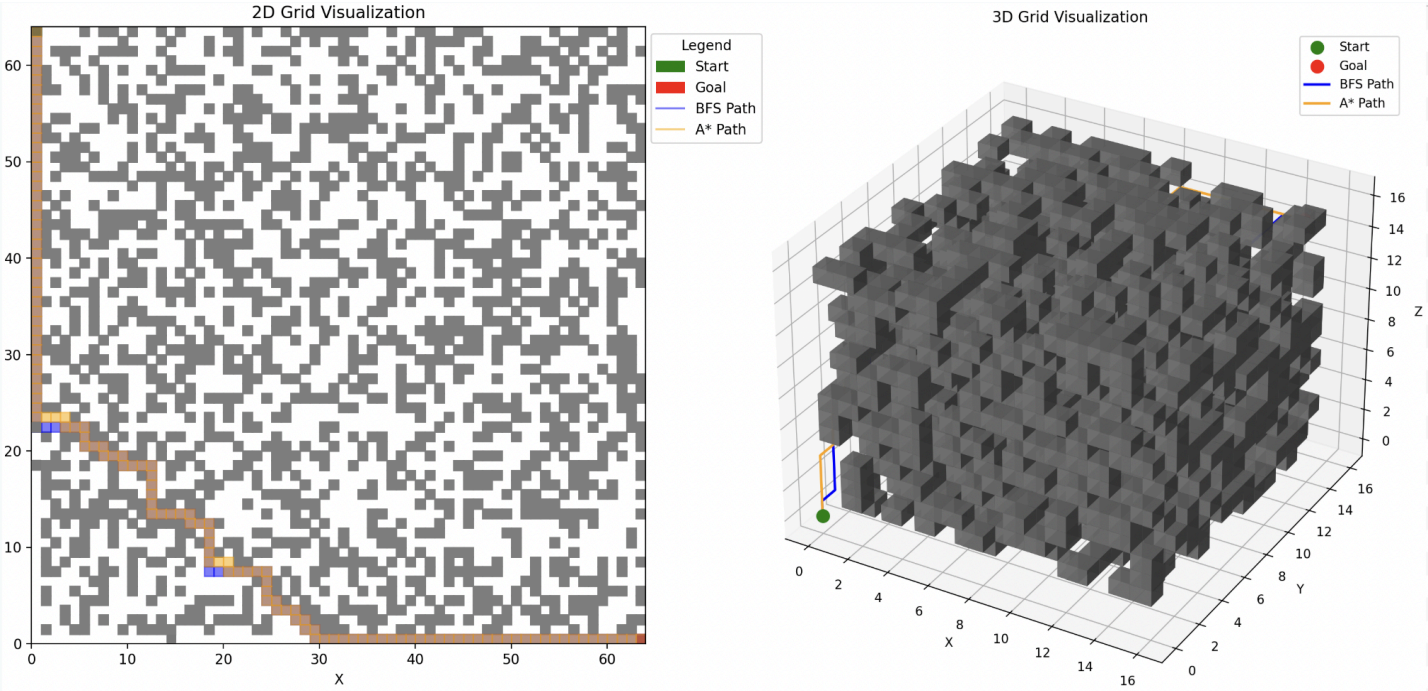


Figure 6: BFS and A* Visualization of Shortest Paths Found in Randomized Grids of 4096 nodes, with 40% Obstacle Densities.

As shown in Figure 6, paths in the 3D grid navigate around obstacles across multiple layers, while the 2D grid remains confined to a single plane. The final reconstructed paths for BFS and A* in both 2D and 3D tend to be very similar.

3.4 Procedure

1. Install Visual Studio Code 1.96.2, and install Python through the application.
2. Copy the provided Python scripts for Breadth-First Search (BFS) and A* algorithms from Appendix A of this essay.
3. Run the program by clicking the ‘Run Python File’ button. Ensure the obstacle density is set to 40% in the Experiment Settings for both 2D and 3D grid environments.

```
# Experiment Settings
GRID_SIZE = 16           # Grid size
OBSTACLE_DENSITY = 0.4   # Obstacle density
CUBE_SIZE = 1            # Size of the obstacle cubes
```

Figure 7: Experimental Settings for 3D algorithms

4. Conduct 5 trials for each grid size in both 2D and 3D for BFS and A*. Record the execution time and number of nodes explored for each trial, and enter the results into a spreadsheet.
5. Calculate the average execution time and number of nodes expanded for each grid size. Using these averages, plot a line graph, and apply a quadratic line of best fit for the 2D algorithms and a cubic line of best fit for the 3D algorithms.

4. Experimental Results

4.1 Data Collection Averages

Grid Size	Breadth First Search		A*	
	2D Time (s)	3D Time (s)	2D Time (s)	3D Time (s)
4096	0.00475	0.00935	0.0012	0.0082
32761	0.03275	0.0673	0.00385	0.0642
110592	0.1165	0.23885	0.0082	0.2334
175561	0.13135	0.3848	0.0112	0.3531
262144	0.2386	0.56095	0.01195	0.54965
1000000	0.8721	2.3298	0.0246	2.4204
2097152	2.23015	4.9329	0.03755	5.60885

Figure 8: Averages (of 5 trials) algorithm execution times.

Grid Size	Breadth First Search		A*	
	2D Nodes Explored	3D Nodes Explored	2D Nodes Explored	3D Nodes Explored
4096	1827	2485	344	1721
32761	13248	19504	1141	13249
110592	46037	66009	2186	45056
175616	52143	105063	3107	68663
262144	94233	156583	3240	107471
1000000	324403	596935	6532	418281
2097152	791591	1252316	9624	896854

Figure 9: Average (of 5 trials) algorithm node exploration

4.2 Data Visualization and Analysis

A log scale has been used for all graphs due to the exponential growth in grid sizes analyzed, which was discussed in Subtopic 3.3. The graphs for BFS and A* in 2 dimensions have been separated into two graphs, due to their vast differences in execution time and number of nodes explored.

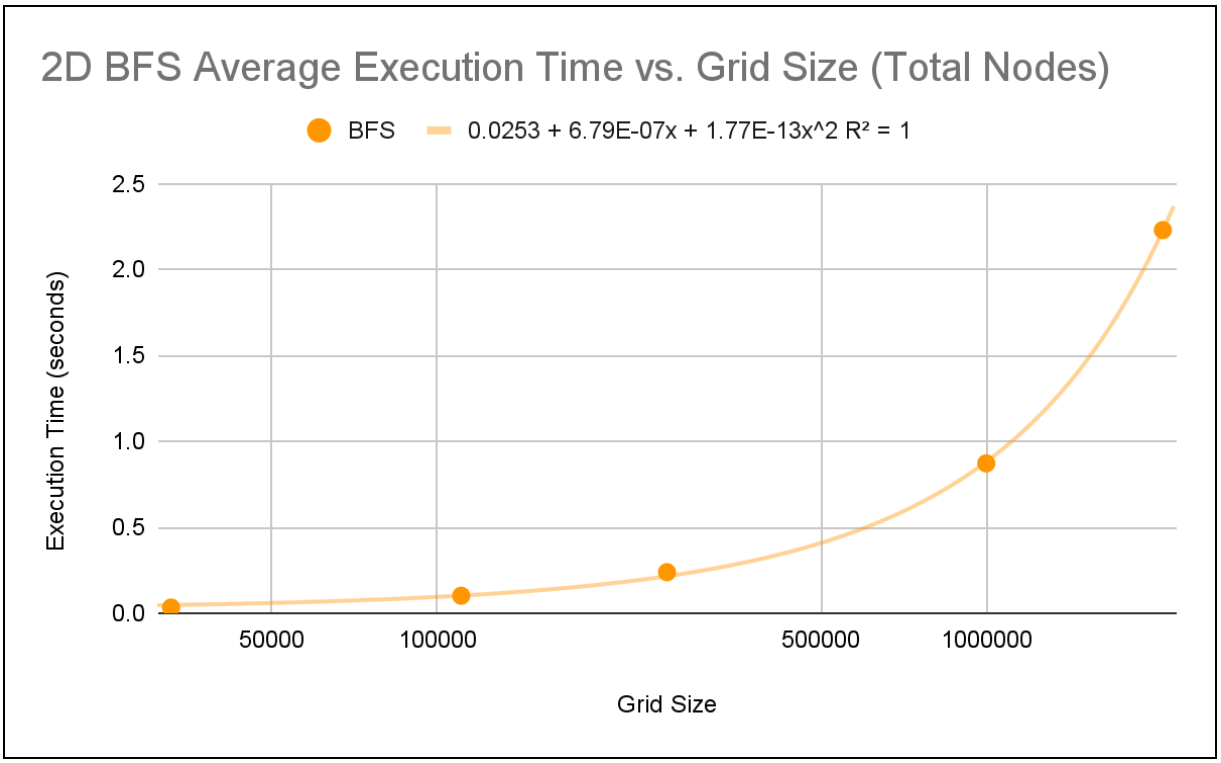


Figure 10: BFS Execution Time Versus Gride Size in 2 Dimensions

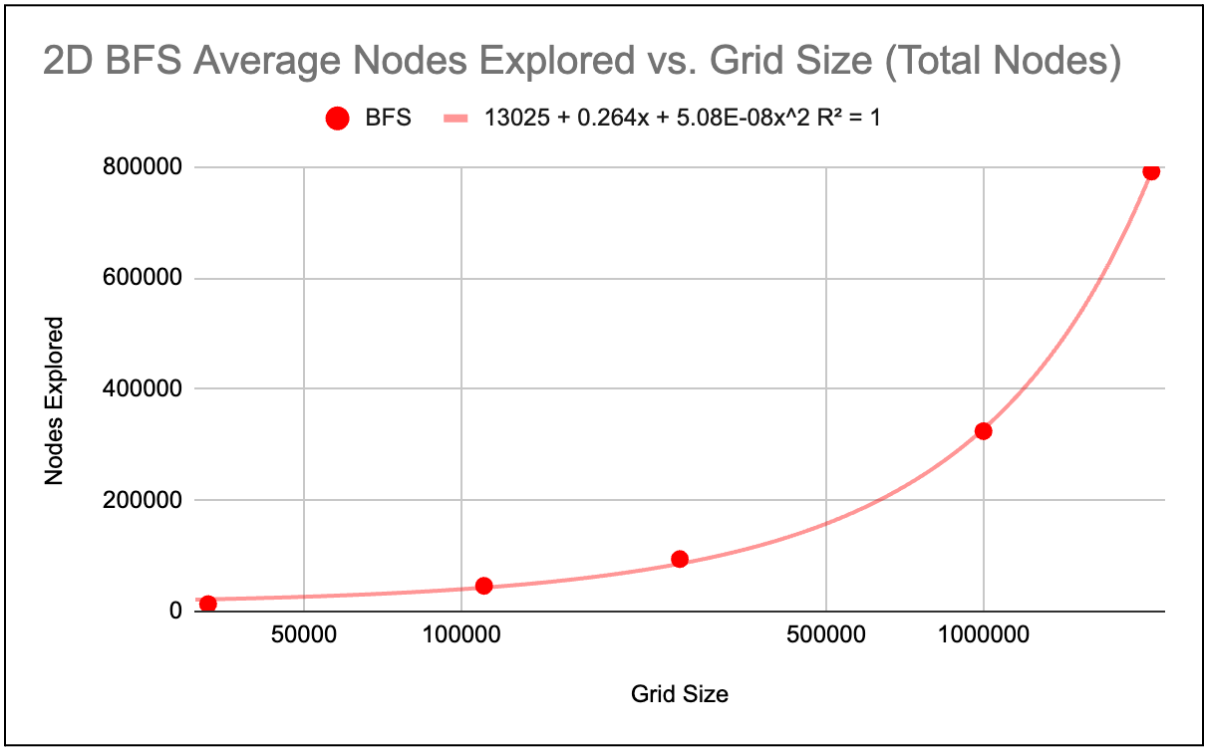


Figure 11: BFS Nodes Explored Versus Grid Size in 2 Dimensions

The quadratic trendlines fitted to Figures 10 and 11, have Pearson Correlation Coefficients $R^2 = 1$, indicating perfect fits to the data. The equation of the trendline for Figure 10, $0.0253 + (6.79 \cdot 10^{-7})x + (1.77 \cdot 10^{-13})x^2$, indicates a quadratic relationship; however, the very small coefficient of the x^2 term $(1.77 \cdot 10^{-13})$ relative to the linear term $(6.79 \cdot 10^{-7})$, suggests that the quadratic growth is not as dominant as expected. This is further supported by Figure 11, which reveals that BFS expands nearly all reachable nodes in a uniform manner. Thus, in a 2D grid where each node has at most four neighbors, the number of edges (E) remains a small multiple of the number of vertices (V).

Figure 11 reinforces this observation, as BFS nodes explored exhibit an almost directly proportional relationship with grid size. This near-linear trend in node exploration suggests that BFS's execution time in Figure 10 is largely dictated by vertex count rather than edges, aligning more closely with $O(V)$ rather than $O(V + E)$. This challenges the assumption that BFS in 2D grids should always exhibit purely quadratic scaling and suggests that its practical growth rate is lower than the theoretical model predicts.

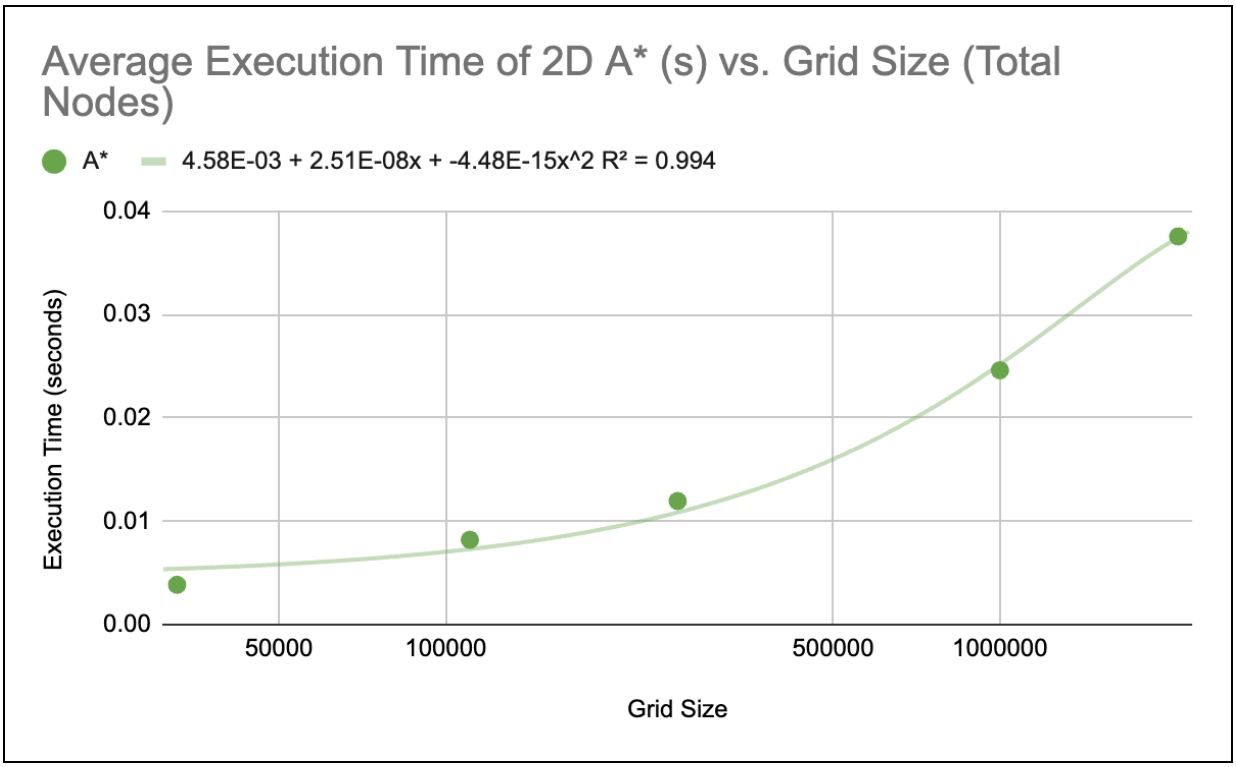


Figure 12: A* Execution Time Versus Grid Size in 2 Dimensions

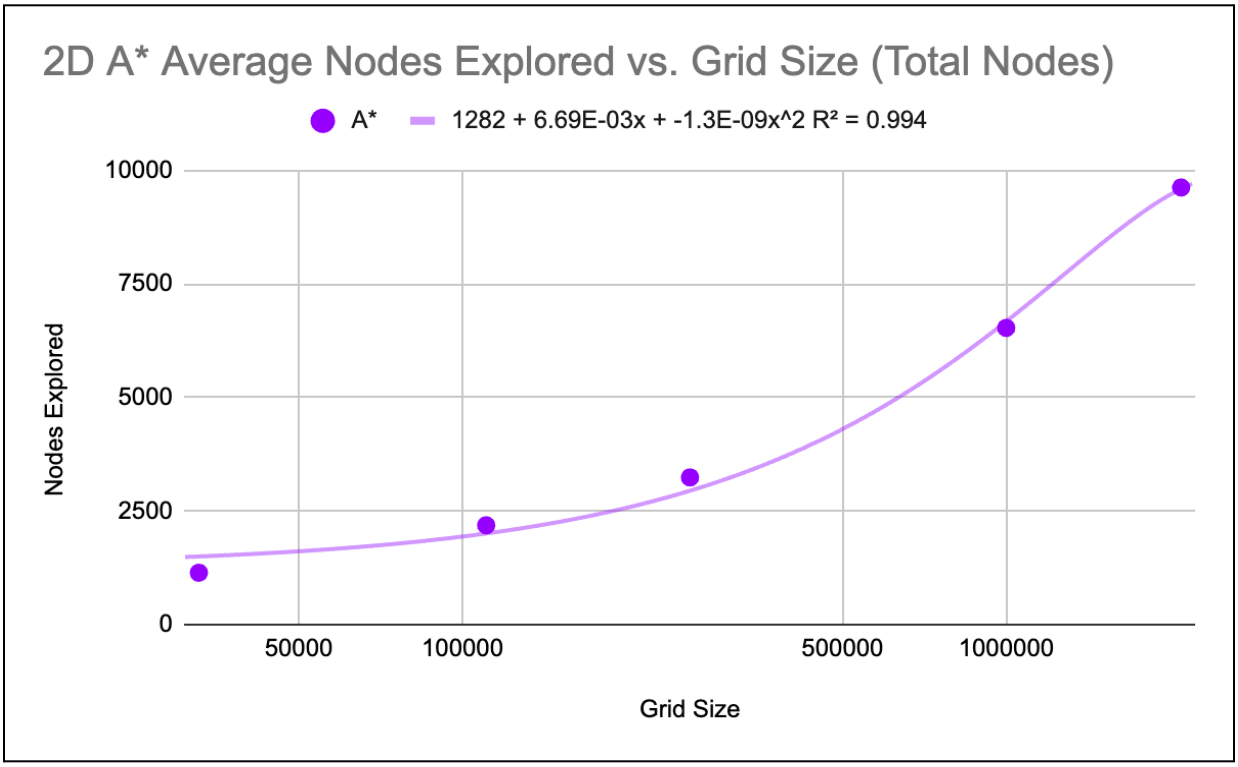


Figure 13: A* Nodes Explored Versus Grid Size in 2 Dimensions

The quadratic trendline fitted to Figure 12, representing A* execution time in 2D, has an R^2 value of 0.994, indicating a strong but slightly imperfect fit. The equation, $4.58 \cdot 10^{-3} + (2.51 \cdot 10^{-8}x) + (-4.48 \cdot 10^{-15})x^2$, shows that the quadratic term $(-4.48 \cdot 10^{-15})$ is not only negligible due to its size but also negative. This is particularly significant because a negative quadratic coefficient implies a slowing growth rate rather than the expected acceleration in execution time. This directly contradicts the original hypothesis, which predicted A* to exhibit a polynomial increase in complexity. Instead of aligning with the theoretical $O(V + E)(log_2 V))$ growth, the graph suggests a near-linear trend, indicating that execution time does not scale quadratically with grid size as expected.

To quantitatively analyze the relative effectiveness of the two algorithms, we can use the difference of their quadratic trend lines of execution time to integrate across the domain of grid sizes evaluated.

$$\int_{4096}^{2097152} (f_{A^*}(x) - f_{BFS}(x))\, dx$$

$$\int_{4096}^{2097152} (0.0253 + (6.79 \cdot 10^{-7})x + (1.77 \cdot 10^{-13})x^2 - (4.58 \cdot 10^{-3} + (2.51 \cdot 10^{-8}x) + (-4.48 \cdot 10^{-15})x^2))\, dx$$

$$\int_{4096}^{2097152} (0.0207 + (6.54 \cdot 10^{-7})x + (1.81 \cdot 10^{-13})x^2)\, dx = 2.04 \cdot 10^6$$

This represents the cumulative efficiency gain, which was found to be $2.04 \cdot 10^6$. This result confirms that A* consistently outperforms BFS in 2D environments, and that this advantage grows as grid size increases. The result provides a numerical measure of A*'s overall time

savings relative to BFS, reinforcing the hypothesis that A* scales more efficiently in 2D grid-based pathfinding. Furthermore, through visual inspection of Figure 11 and Figure 13, it is clear that A* outperforms BFS by exploring less nodes.

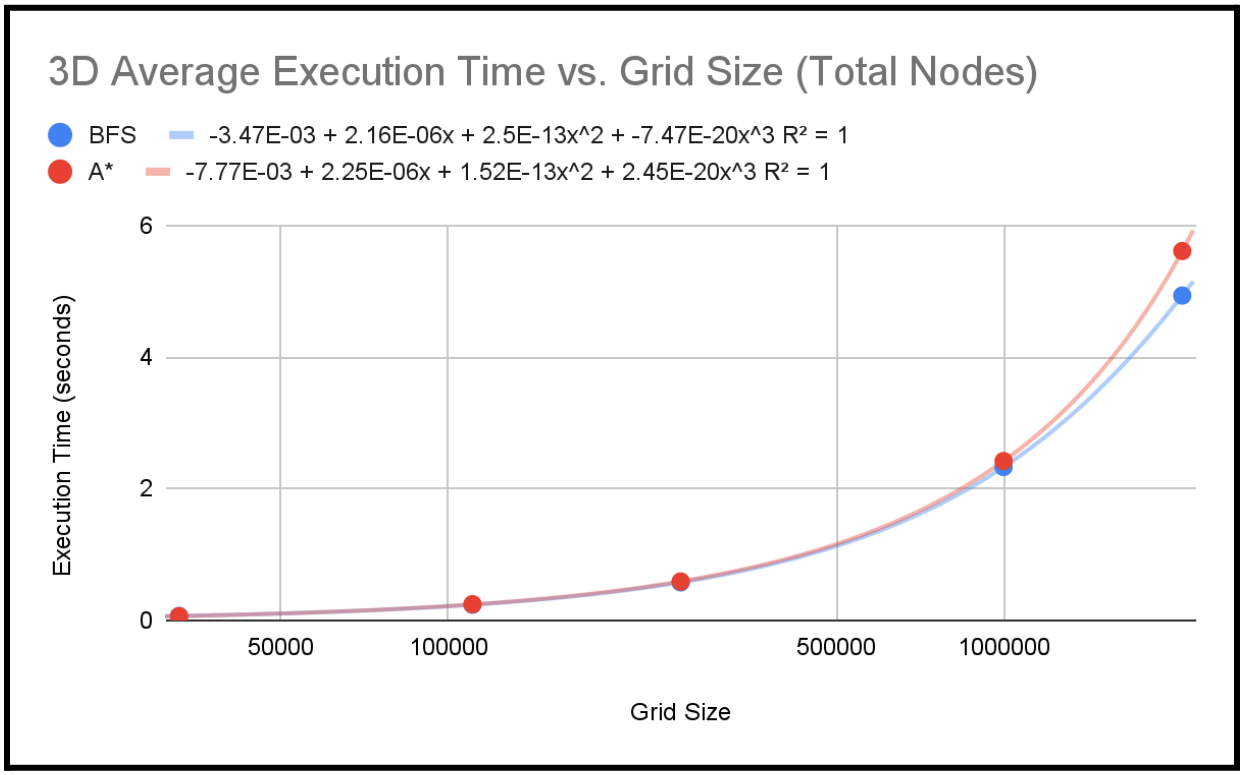


Figure 14: BFS and A* Execution Times Versus Grid Size in 3 Dimensions

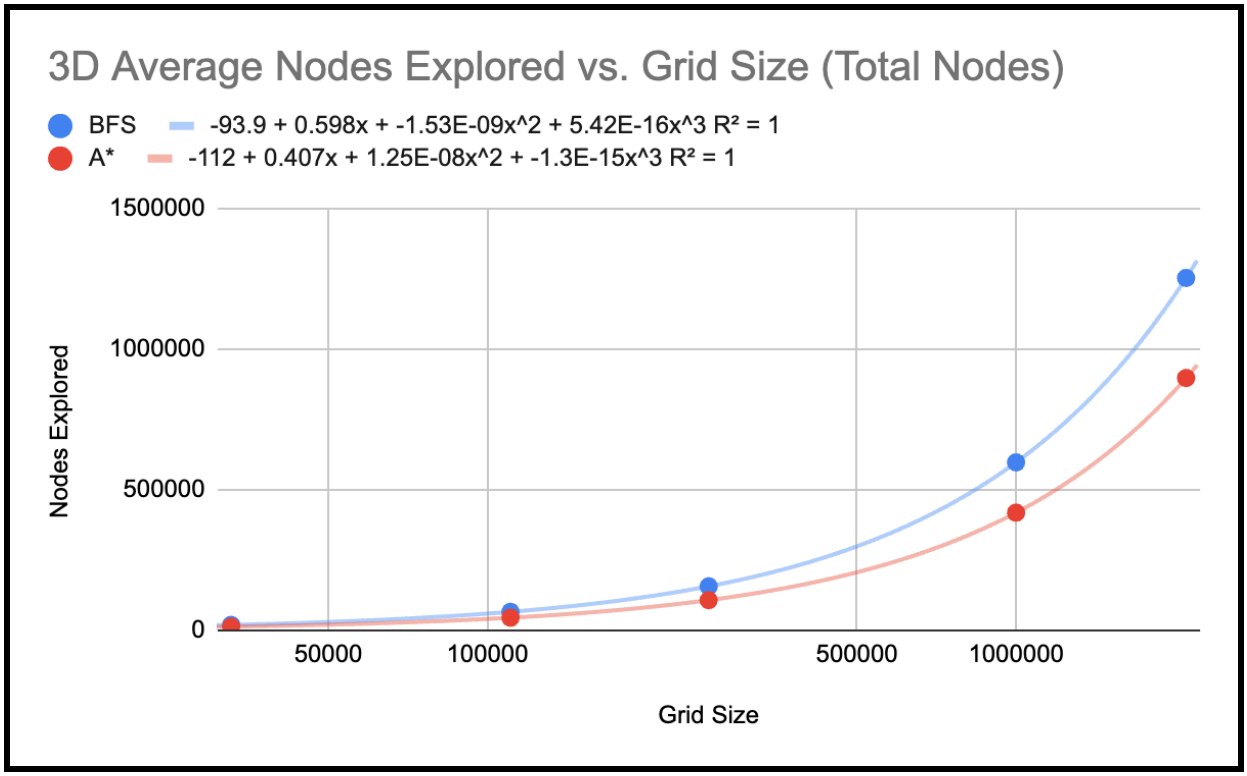


Figure 15: BFS and A* Nodes Explored Versus Grid Size in 3 Dimensions

The trendlines in Figure 14 both have an R^2 value of 1 for both BFS and A*, indicating a perfect fit to the data. However, unlike in 2D, where A* consistently outperformed BFS, the 3D results show nearly identical execution times, contradicting the hypothesis that A* would maintain a significant advantage in three dimensions. However, when considering Figure 15, the results suggest a tradeoff between search effort and computational overhead. BFS consistently explored more nodes than A*, yet its execution time remains smaller at large grid sizes.

5. Evaluation

A* consistently outperformed BFS in 2D by exploring far fewer nodes, resulting in shorter execution times, as shown in Figures 10 and 12. As grid sizes grow, A*'s ability to selectively focus on promising paths results in significantly fewer nodes visited and lower execution times compared to BFS.

However, this advantage diminishes in 3D. Although A* still explores fewer nodes than BFS (Figure 15), its execution time becomes nearly identical to it (Figure 14). This shift highlights how the added computational overhead of managing A*'s priority queue counterbalances the benefits of its heuristic. The increased branching factor in 3D, rising from 4 possible directions per node in 2D to 6 in 3D, contributes to this trend. BFS benefits from rapid outward expansion, while A*'s heuristic, effective in 2D, struggles to maintain the same efficiency in higher-dimensional search spaces.

Another contributing factor is the heuristic itself. The Manhattan distance heuristic, which works well in 2D grids, becomes less effective in 3D, where movement occurs along three axes. This reduced heuristic accuracy forces A* to perform more unnecessary calculations, further limiting its advantage. Despite this, A* still retains its core strength of minimizing search effort by exploring fewer nodes. However, in 3D, this reduced search effort no longer translates into a clear execution time benefit, indicating that as dimensionality increases, the choice between BFS and A* depends less on speed and more on search effort and memory efficiency.

These findings highlight the role of dimensionality in determining algorithm efficiency. In lower dimensions, where the search space and branching factor are smaller, A* demonstrates a clear advantage in both execution time and node exploration. However, in higher dimensions, heuristic inefficiencies and computational overhead reduce its practical benefits. This suggests that in high-dimensional pathfinding applications, the decision between BFS and A* may depend less on execution time and more on factors such as memory constraints and search effort. A* remains the preferable choice when minimizing node exploration is critical, but the diminishing time advantage in higher dimensions calls into question whether its use is always justified, especially as grid sizes continue to scale.

5.1 Experiment's Implications for Further Dimensions

As dimensionality increases, the exponential growth of the search space presents challenges, particularly in balancing computational cost and efficiency. While the results in 3D suggest that A* and BFS have similar execution times in large grid sizes, the significant reduction in nodes explored by A* indicates that its performance advantage lies in its ability to constrain unnecessary exploration. In higher dimensions, this distinction would likely become more pronounced, as the number of nodes and branching factor would amplify the inefficiencies of BFS.

In 4D and beyond, scenarios with extremely sparse obstacles or uniform environments might reduce the effectiveness of A*'s heuristic, potentially narrowing the performance gap between the two algorithms. However, as environments become more complex, A*'s ability to

strategically prioritize nodes would likely make it more reliable in navigating through multidimensional spaces efficiently. BFS, by contrast, would suffer not only from an exponential increase in execution time but also from memory limitations.

This experiment demonstrates the need for possible hybrid approaches or heuristic optimizations in higher dimensions, particularly for algorithms like A*, to mitigate the rising computational and memory costs. It also highlights the importance of understanding the trade-offs between exhaustive and heuristic-based searches, as these trade-offs will increasingly dictate the feasibility of algorithms in higher-dimensional tasks. For applications such as 4D robotics or data analysis in high-dimensional spaces, the insights from this study suggest that while A* remains effective, further refinements may be necessary to maintain scalability and efficiency.

5.2 Limitations

The grids used in this experiment were randomly generated with a fixed obstacle density of 40%, ensuring consistency across trials. However, this uniform density creates a particular type of search space that may inherently favor or disadvantage certain algorithms. For example, a lower density would result in more open spaces, reducing the importance of heuristic guidance, while a higher density could create bottlenecks where informed search methods like A* would have more opportunity to demonstrate their advantage. This highlights that obstacle density directly shapes the challenge each algorithm faces, influencing both search effort and execution time.

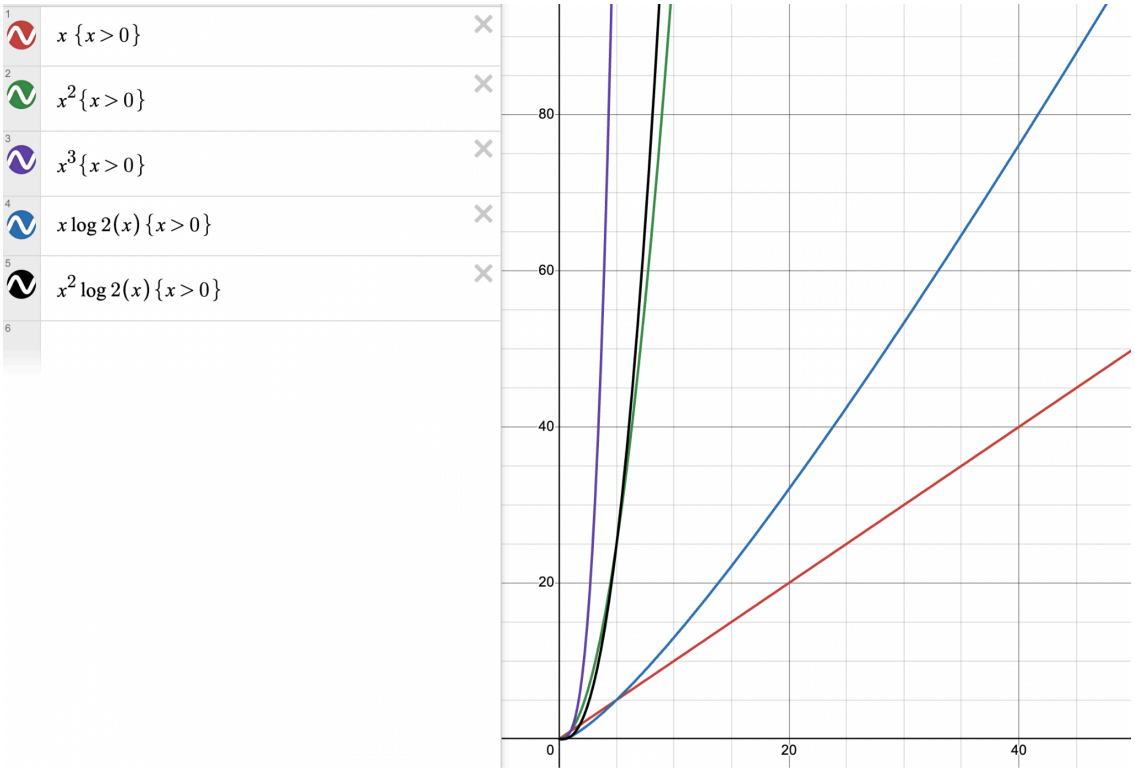


Figure 16: Graphs of theoretical time complexities with graph of $y = x$ (Desmos)

Figure 16 compares the graph of $y = x$ with possible time complexities for BFS and A* in 2D and 3D. It reveals noticeable curvature at very small grid sizes, but as grid size increases, both functions exhibit quasi-linear behavior. This explains the small coefficients seen in the quadratic and cubic terms from the experimental trendlines, indicating that the theoretical polynomial complexity, while present, may be suppressed for the grid sizes tested. This suggests that the practical time complexity for BFS and A* in grid-based environments may appear closer to linear at experimentally feasible grid sizes, even though theoretical analysis predicts polynomial growth.

Furthermore, the experiment focused on 2D and 3D grid environments, thus the results do not account for performance in higher dimensions. While it can be inferred that A*'s heuristic will continue to mitigate some of the computational costs associated with higher dimensions, the

algorithms were not explicitly tested in these scenarios, limiting the generalizability of the findings to truly multidimensional environments.

The hardware used (MacBook Air M3) was not optimized for computationally intensive tasks. BFS, in particular, is memory-intensive, and the available RAM may have constrained its performance on larger grids. A*'s reliance on priority queues may also have been affected by the hardware's processing power. Background processes and Python's overhead could have further introduced variability. Future research could address these limitations by testing larger grids, higher dimensions, and using high-performance hardware.

6. Conclusion

This study demonstrates how increasing dimensionality from 2D to 3D directly impacts the performance of BFS and A*, with both algorithms experiencing increased execution times and greater node exploration. The results highlight that neither algorithm universally outperforms the other; BFS benefits from its simplicity, while A* reduces unnecessary exploration through its heuristic. Crucially, this investigation emphasizes the importance of context when choosing a pathfinding algorithm. In smaller, lower-dimensional grids, A* maintains clear advantages by limiting search effort. However, in larger 3D grids, its computational overhead becomes a limiting factor, closing the performance gap with BFS. This suggests that for pathfinding in higher dimensions, careful consideration must be given to whether minimizing search effort or prioritizing execution time is more critical to the application.

While this experiment was limited to 2D and 3D, its insights lay the groundwork for future research into pathfinding in higher dimensions. As real-world applications like robotics, simulations, and scientific modeling increasingly engage with multidimensional data, understanding how these algorithms scale will be essential for developing efficient and adaptable solutions.

7. Bibliography

Cox, Graham. “A* Pathfinding Algorithm.” Baeldung on Computer Science, March 18, 2024. https://www.baeldung.com/cs/a-star-algorithm. Accessed 10 July 2024.

DataCamp. “Big O Notation and Time Complexity Guide: Intuition and Math.” DataCamp, July 8, 2024. https://www.datacamp.com/tutorial/big-o-notation-time-complexity. Accessed 12 July 2024.

Desmos. Desmos Graphing Calculator. Amherst: Desmos, June 30, 2011. https://www.desmos.com/calculator. Accessed January 11, 2025.

GeeksforGeeks. “Breadth First Search or BFS for a Graph.” Last modified March 6, 2024. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. Accessed July 15, 2024.

Huang, Shen. “What is Big O Notation Explained: Space and Time Complexity.” freeCodeCamp, January 16, 2020. https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/. Accessed July 11, 2024.

Johnson, Amit. “Heuristics.” Amit’s Thoughts on Pathfinding, 1997–2024. https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html. Accessed July 28, 2024.

Luo, Yuan, Jiakai Lu, Yi Zhang, Qiong Qin, and Yanyu Liu. 2022. "3D JPS Path Optimization Algorithm and Dynamic-Obstacle Avoidance Design Based on Near-Ground Search Drone" Applied Sciences 12, no. 14: 7333. https://doi.org/10.3390/app12147333. Accessed October 11, 2024.

Nawaf Hazim, Sinan Sameer Mahmood Al-Dabbagh, and Mustafa Abdul Sahib Naser. “Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm.” Journal of Computer and Communications 4, no. 11 (2016): 15-25. https://www.scirp.org/journal/paperinformation?paperid=70460#f1. Accessed July 30, 2024.

OpenAI. ChatGPT-4o: AI Language Model. OpenAI, 2024. https://openai.com. Accessed October 22, 2024.

Östberg, Oscar. “A Comparison of Algorithms for the Purpose of Path-Finding in a 3D Grid.” Bachelor’s thesis, Blekinge Institute of Technology, May 2021. https://www.diva-portal.org/smash/get/diva2:1639201/FULLTEXT02.pdf. Accessed August 2, 2024.

Pathfindout. “Pathfinding Visualizer: Simulating Pathfinding in 2D.” Accessed October 2024. https://pathfindout.com/. Accessed July 16, 2024.

Tong, Xin, and Zhenyi Tang. “BFS & DFS.” CS 225, University of Illinois at Urbana-Champaign, Spring 2023. https://courses.grainger.illinois.edu/cs225/sp2023/resources/bfs-dfs/. Accessed July 20, 2024.

Yap, Peter. “Grid-Based Path-Finding.” In AI 2022: Advances in Artificial Intelligence, edited by R. Cohen and B. Spencer, 44–55. LNAI 2338. Berlin: Springer-Verlag, 2022. https://svn.sable.mcgill.ca/sable/courses/COMP763/oldpapers/yap-02-grid-based.pdf. Accessed August 5, 2024.

8. Appendix

Appendix A: Execution Times and Node Exploration Data Collection

Algorithm Execution Times					
		2 Dimensions		3 Dimensions	
Grid Size	**Trial No.**	**BFS Execution Time (s)**	**A* Execution Time (s)**	**BFS Execution Time (s)**	**A* Execution Time (s)**
$64^2 = 16^3$	1	0.0042	0.0008	0.0092	0.0083
	2	0.0036	0.0009	0.0091	0.0070
	3	0.0029	0.0008	0.0090	0.0078
	4	0.0045	0.0013	0.0089	0.0083
	5	0.0053	0.0016	0.0095	0.0081
$181^2 \approx 32^3$	1	0.0339	0.0039	0.0674	0.0648
	2	0.0317	0.0040	0.0675	0.0644
	3	0.0231	0.0029	0.0680	0.0537
	4	0.0199	0.0043	0.0675	0.0598
	5	0.0316	0.0038	0.0672	0.0636
$333^2 \approx 48^3$	1	0.1078	0.0073	0.2395	0.2346
	2	0.0786	0.0073	0.2408	0.2266
	3	0.1019	0.0073	0.2400	0.2346
	4	0.0825	0.0089	0.2415	0.2254
	5	0.1252	0.0091	0.2382	0.2322
$419^2 \approx 56^3$	1	0.1202	0.0116	0.3773	0.3460
	2	0.1205	0.0086	0.3818	0.3640
	3	0.1410	0.0109	0.3895	0.3806
	4	0.1444	0.0099	0.3877	0.3618
	5	0.1425	0.0108	0.3923	0.3602

Algorithm Execution Times					
		2 Dimensions		3 Dimensions	
Grid Size	**Trial No.**	**BFS Execution Time (s)**	**A* Execution Time (s)**	**BFS Execution Time (s)**	**A* Execution Time (s)**
$512^2 = 64^3$	1	0.2444	0.0114	0.5620	0.5593
	2	0.2156	0.0110	0.5567	0.5253
	3	0.1794	0.0148	0.5582	0.5519
	4	0.1927	0.0119	0.5596	0.5317
	5	0.2328	0.0125	0.5599	0.5400

Algorithm Node Exploration					
		2 Dimensions		3 Dimensions	
Grid Size	**Trial No.**	**BFS Nodes Explored**	**A* Nodes Explored**	**BFS Nodes Explored**	**A* Nodes Explored**
$64^2 = 16^3$	1	1602	242	2484	1722
	2	1325	274	2461	1486
	3	1074	270	2438	1607
	4	1723	363	2400	1774
	5	2052	447	2487	1720
$181^2 \approx 32^3$	1	13742	1159	19500	13225
	2	12664	1163	19556	13411
	3	9167	886	19575	11162
	4	7850	1215	19513	12576
	5	12754	1124	19509	13274

Algorithm Node Exploration					
		2 Dimensions		3 Dimensions	
Grid Size	**Trial No.**	**BFS Nodes Explored**	**A* Nodes Explored**	**BFS Nodes Explored**	**A* Nodes Explored**
$333^2 \approx 48^3$	1	43178	1994	65967	44842
	2	31576	2062	66033	42791
	3	41512	2117	65931	44171
	4	33358	2430	65932	42592
	5	48895	2379	66051	45271
$419^2 \approx 56^3$	1	47678	3249	104906	68284
	2	47567	2277	104917	70888
	3	57035	2986	105039	73744
	4	56905	2759	104649	69762
	5	56607	2966	105221	69042
$512^2 = 64^3$	1	97195	3152	156749	109515
	2	85796	3008	156607	102889
	3	71311	4048	156061	107298
	4	76474	3274	156447	103683
	5	91271	3328	156417	105427

Appendix B: Code Generation

To construct the pseudocode examples in Figure 1 and 4, as well as the Python code for the experiment itself, I used the Large Language Model ChatGPT-4o. The necessary code was generated through the use of prompting. Below are the main prompts that were used to generate the code.

Prompt	**Explanation**
Write pseudocode for BFS in a 2D grid. The pseudocode should include clear initialization, a main loop that explores neighboring nodes in 4 directions (up, down, left, right), and should guarantee that the shortest path is found. Use a queue for the open list, a set to track visited nodes, and a parent map to reconstruct the final path. Ensure the pseudocode is commented to explain each section.	This prompt provided the basis for **Figure 1**, the BFS pseudocode. By specifying the 2D grid context, the 4-connected neighborhood, and the need to reconstruct the path, the generated pseudocode aligned directly with my EE's experiment. The comments ensure that it could later be adapted into explanations within the essay itself.
Write pseudocode for A* search in a 2D grid. The pseudocode should include clear initialization, using a priority queue for the open list, and a cost tracking system (gScore and fScore). The heuristic function should use Manhattan distance. The algorithm should repeatedly expand the node with the lowest fScore, updating costs and reconstructing the final path when the goal is reached. Comment each section to explain its purpose.	This prompt generated **Figure 4**, the A* pseudocode. The requirement for a priority queue, cost tracking, and Manhattan heuristic directly reflects the algorithm's theoretical basis covered in my background section. The comments also ensured that the pseudocode was accessible for explanation.
Write Python code to implement both BFS and A* pathfinding algorithms in a 2D grid. The grid should have randomly placed obstacles with a specified obstacle density, and guaranteed connectivity between the start and goal. BFS should use a queue, and A* should use a priority queue with Manhattan distance heuristic. For both algorithms, track and return execution time, path length, and number of nodes expanded. The code should be structured to allow easy visualization in matplotlib, and should be formatted to ensure clarity and readability. Comment each section appropriately.	This prompt produced the initial version of the 2D BFS and A* Python code used in my experiment. It ensured the algorithms aligned directly with the theoretical versions discussed in my essay, and the focus on execution time, nodes expanded, and path length ensured it supported my data analysis. It also enabled visualization, which was helpful for debugging and understanding the grid environments.

Write Python code to implement BFS and A* pathfinding in a 3D grid. The 3D grid should have obstacles placed with a specified density, and should guarantee a path exists from the start (0,0,0) to the goal (N-1, N-1, N-1). BFS should expand nodes in 6 directions, and A* should use Manhattan distance generalized to 3D. Track execution time, path length, and number of nodes expanded for both algorithms. Structure the code to allow visualization in matplotlib's 3D plotting library. Comment all parts to ensure clarity.	This prompt generated the initial version of the 3D BFS and A* Python code. It introduced the 3D grid environment, requiring adjustment of the neighborhood connectivity and visualization process. By keeping the same metrics (time, length, nodes expanded), it ensured direct comparability between 2D and 3D results, supporting the dimensionality focus of my EE.

Appendix C: Python Code

The following blocks of code can be used in Python 1.96.2 to recreate the experiments. The visualization methods have been "commented" out of the code, since it uses a lot of memory and affects the core experiment. To use the visualization tools, remove the "##" in front of method "def visualize_2d_path" where it is defined and where it is called at the end. The same can be done for the 3D algorithms. As stated earlier, the experimental variables can be adjusted under "# Experiment Settings".

2D BFS and A* Code

```
import time
import heapq
import random
import matplotlib.pyplot as plt

# Experiment Settings
GRID_SIZE = 64         # Grid size
OBSTACLE_DENSITY = 0.4  # Obstacle density

def generate_2d_grid_with_path(size, obstacle_probability=0.4):
    """
    Generates a 2D grid with obstacles, ensuring a path exists between start and goal.
    """
    random.seed(time.time())  # Ensure a randomized environment each run

    grid = [[False for _ in range(size)] for _ in range(size)]
    for x in range(size):
```

```
        for y in range(size):
            if random.random() < obstacle_probability:
                grid[x][y] = True

    # Ensure start and goal are free
    grid[0][0] = False
    grid[size - 1][size - 1] = False

    # Carve a simple path to guarantee connectivity
    x, y = 0, 0
    while (x, y) != (size - 1, size - 1):
        grid[x][y] = False
        if x < size - 1:
            x += 1
        elif y < size - 1:
            y += 1

    return grid

def is_valid_2d(grid, x, y):
    size = len(grid)
    return 0 <= x < size and 0 <= y < size and not grid[x][y]

def get_neighbors_4_connected(x, y):
    """
    Returns neighbors in a randomized order to introduce variability for BFS.
    """
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    random.shuffle(directions)  # Randomize the order of neighbors
    for dx, dy in directions:
        yield x + dx, y + dy

def bfs_2d_with_metrics(grid, start, goal):
    """
    BFS with randomized neighbor exploration.
    """
    start_time = time.time()
    from collections import deque
    queue = deque([start])
    visited = set([start])
    parent = {start: None}
    nodes_expanded = 0
```

```
    while queue:
        current = queue.popleft()
        nodes_expanded += 1
        if current == goal:
            path = []
            node = current
            while node is not None:
                path.append(node)
                node = parent[node]
            path.reverse()
            end_time = time.time()
            return len(path), end_time - start_time, nodes_expanded, path

        x, y = current
        for nx, ny in get_neighbors_4_connected(x, y):
            if is_valid_2d(grid, nx, ny) and (nx, ny) not in visited:
                visited.add((nx, ny))
                parent[(nx, ny)] = current
                queue.append((nx, ny))

    end_time = time.time()
    return None, end_time - start_time, nodes_expanded

def heuristic_manhattan_2d(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star_2d_with_metrics(grid, start, goal):
    start_time = time.time()
    open_set = []
    heapq.heappush(open_set, (0, start))
    g_score = {start: 0}
    came_from = {start: None}
    nodes_expanded = 0

    while open_set:

        _, current = heapq.heappop(open_set)
        nodes_expanded += 1

        if current == goal:
            path = []
```

```
            node = current
            while node is not None:
                path.append(node)
                node = came_from[node]
            path.reverse()
            end_time = time.time()
            return len(path), end_time - start_time, nodes_expanded, path

        x, y = current
        current_g = g_score[current]

        for nx, ny in get_neighbors_4_connected(x, y):
            if not is_valid_2d(grid, nx, ny):
                continue
            tentative_g = current_g + 1
            if tentative_g < g_score.get((nx, ny), float('inf')):
                came_from[(nx, ny)] = current
                g_score[(nx, ny)] = tentative_g
                f_score = tentative_g + heuristic_manhattan_2d((nx, ny), goal)
                heapq.heappush(open_set, (f_score, (nx, ny)))

    end_time = time.time()
    return None, end_time - start_time, nodes_expanded, None

## def visualize_2d_path(grid, start, goal, path_bfs=None, path_astar=None):
    """
    Visualizes a 2D grid, obstacles, and paths (if provided) in a matplotlib figure.
    """
    size = len(grid)
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_title("2D Grid Visualization")
    ax.set_xlabel('X')
    ax.set_ylabel('Y')

    # Plot the grid and obstacles
    for x in range(size):
        for y in range(size):
            if grid[x][y]:
                ax.add_patch(plt.Rectangle((y, size - x - 1), 1, 1, color='gray'))

    # Plot the start and goal nodes
```

```
    ax.add_patch(plt.Rectangle((start[1], size - start[0] - 1), 1, 1, color='green',
label='Start'))
    ax.add_patch(plt.Rectangle((goal[1], size - goal[0] - 1), 1, 1, color='red',
label='Goal'))

    # Plot BFS path if provided
    if path_bfs is not None:
        for x, y in path_bfs:
            ax.add_patch(plt.Rectangle((y, size - x - 1), 1, 1, color='blue',
alpha=0.5))
        ax.plot([], [], color='blue', alpha=0.5, label='BFS Path')  # Add to legend

    # Plot A* path if provided
    if path_astar is not None:
        for x, y in path_astar:
            ax.add_patch(plt.Rectangle((y, size - x - 1), 1, 1, color='orange',
alpha=0.5))
        ax.plot([], [], color='orange', alpha=0.5, label='A* Path')  # Add to legend

    ax.set_xlim(0, size)
    ax.set_ylim(0, size)
    ax.legend(loc='upper right', bbox_to_anchor=(1.2, 1), title="Legend")  # Adjust
legend position
    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()

if __name__ == "__main__":
    # Fixed Start and Goal Nodes
    start_node = (0, 0)
    goal_node = (GRID_SIZE - 1, GRID_SIZE - 1)

    # Generate a grid with guaranteed connectivity
    grid = generate_2d_grid_with_path(GRID_SIZE, obstacle_probability=OBSTACLE_DENSITY)

    ## print(f"Start: {start_node}, Goal: {goal_node}")

    # Run BFS
    bfs_result = bfs_2d_with_metrics(grid, start_node, goal_node)
    if bfs_result:
        bfs_path_length, bfs_time, bfs_nodes, bfs_path = bfs_result
        print(f"BFS - Path Length: {bfs_path_length}, Time: {bfs_time:.4f}s, Nodes
Expanded: {bfs_nodes}")
```

```
    else:
        print("BFS - No path found")
        bfs_path = None

    # Run A*
    a_star_result = a_star_2d_with_metrics(grid, start_node, goal_node)
    if a_star_result:
        a_star_path_length, a_star_time, a_star_nodes, a_star_path = a_star_result
        print(f"A* - Path Length: {a_star_path_length}, Time: {a_star_time:.4f}s, Nodes
Expanded: {a_star_nodes}")
    else:
        print("A* - No path found")
        a_star_path = None

    # Visualize the results
    ## visualize_2d_path(grid, start_node, goal_node, path_bfs=bfs_path,
path_astar=a_star_path)
```

3D BFS and A* Code

```
import time
import heapq
import random
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

# Experiment Settings
GRID_SIZE = 64          # Grid size
OBSTACLE_DENSITY = 0.4  # Obstacle density
CUBE_SIZE = 1           # Size of the obstacle cubes

def generate_3d_grid_with_path(size, obstacle_probability=0.4):
    random.seed(time.time())  # Ensure a randomized environment each run

    grid = [[[False for _ in range(size)] for _ in range(size)] for _ in range(size)]
    for x in range(size):
        for y in range(size):
            for z in range(size):
                if random.random() < obstacle_probability:
                    grid[x][y][z] = True
```

```
    # Ensure start and goal are free
    grid[0][0][0] = False
    grid[size - 1][size - 1][size - 1] = False

    # Carve a simple path to guarantee connectivity
    x, y, z = 0, 0, 0
    while (x, y, z) != (size - 1, size - 1, size - 1):
        grid[x][y][z] = False
        if x < size - 1:
            x += 1
        elif y < size - 1:
            y += 1
        elif z < size - 1:
            z += 1

    return grid

def is_valid(grid, x, y, z):
    size = len(grid)
    if 0 <= x < size and 0 <= y < size and 0 <= z < size:
        return not grid[x][y][z]
    return False

def get_neighbors_6_connected(x, y, z):
    """
    Returns neighbors in a randomized order to introduce variability for BFS.
    """
    directions = [
        (1, 0, 0), (-1, 0, 0),
        (0, 1, 0), (0, -1, 0),
        (0, 0, 1), (0, 0, -1)
    ]
    random.shuffle(directions)  # Randomize the order of neighbors
    for dx, dy, dz in directions:
        yield x + dx, y + dy, z + dz

def bfs_3d_with_metrics(grid, start, goal):
    """
    BFS with randomized neighbor exploration.
    """
    start_time = time.time()
    from collections import deque
```

```
    queue = deque([start])
    visited = set([start])
    parent = {start: None}
    nodes_expanded = 0

    while queue:

        current = queue.popleft()
        nodes_expanded += 1
        if current == goal:
            path = []
            node = current
            while node is not None:
                path.append(node)
                node = parent[node]
            path.reverse()
            end_time = time.time()
            return len(path), end_time - start_time, nodes_expanded, path

        x, y, z = current
        for nx, ny, nz in get_neighbors_6_connected(x, y, z):
            if is_valid(grid, nx, ny, nz) and (nx, ny, nz) not in visited:
                visited.add((nx, ny, nz))
                parent[(nx, ny, nz)] = current
                queue.append((nx, ny, nz))

    end_time = time.time()
    return None, end_time - start_time, nodes_expanded, None

def heuristic_manhattan_3d(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1]) + abs(a[2] - b[2])

def a_star_3d_with_metrics(grid, start, goal):
    start_time = time.time()
    open_set = []
    heapq.heappush(open_set, (0, start))
    g_score = {start: 0}
    came_from = {start: None}
    nodes_expanded = 0

    while open_set:
```

```
        _, current = heapq.heappop(open_set)
        nodes_expanded += 1

        if current == goal:
            path = []
            node = current
            while node is not None:
                path.append(node)
                node = came_from[node]
            path.reverse()
            end_time = time.time()
            return len(path), end_time - start_time, nodes_expanded, path

        x, y, z = current
        current_g = g_score[current]

        for nx, ny, nz in get_neighbors_6_connected(x, y, z):
            if not is_valid(grid, nx, ny, nz):
                continue
            tentative_g = current_g + 1
            if tentative_g < g_score.get((nx, ny, nz), float('inf')):
                came_from[(nx, ny, nz)] = current
                g_score[(nx, ny, nz)] = tentative_g
                f_score = tentative_g + heuristic_manhattan_3d((nx, ny, nz), goal)
                heapq.heappush(open_set, (f_score, (nx, ny, nz)))

    end_time = time.time()
    return None, end_time - start_time, nodes_expanded, None

##def visualize_3d_path(grid, start, goal, path_bfs=None, path_astar=None):
    """
    Visualizes a 3D grid, obstacles as 3D cubes, and paths (if provided) in a 3D
matplotlib figure.
    """
    size = len(grid)

    # Collect obstacle coordinates for plotting
    obs_x = []
    obs_y = []
    obs_z = []
    for x in range(size):
        for y in range(size):
```

```
            for z in range(size):
                if grid[x][y][z]:
                    obs_x.append(x)
                    obs_y.append(y)
                    obs_z.append(z)

    # Create the figure and axes
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title("3D Grid Visualization")
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    # Plot obstacles as cubes
    for x, y, z in zip(obs_x, obs_y, obs_z):
        ax.bar3d(x, y, z, CUBE_SIZE, CUBE_SIZE, CUBE_SIZE, shade=True, color='gray',
alpha=0.8)

    # Plot start and goal nodes
    ax.scatter(start[0], start[1], start[2], c='g', marker='o', s=100, label='Start')
    ax.scatter(goal[0], goal[1], goal[2], c='r', marker='o', s=100, label='Goal')

    # Plot BFS path if provided
    if path_bfs is not None:
        px, py, pz = zip(*path_bfs)
        ax.plot(px, py, pz, c='blue', linewidth=2, label='BFS Path')

    # Plot A* path if provided
    if path_astar is not None:
        px, py, pz = zip(*path_astar)
        ax.plot(px, py, pz, c='orange', linewidth=2, label='A* Path')

    ax.legend()
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    # Fixed Start and Goal Nodes
    start_node = (0, 0, 0)
    goal_node = (GRID_SIZE - 1, GRID_SIZE - 1, GRID_SIZE - 1)
```

```
    # Generate a grid with guaranteed connectivity
    grid = generate_3d_grid_with_path(GRID_SIZE, obstacle_probability=OBSTACLE_DENSITY)

    ## print(f"Start: {start_node}, Goal: {goal_node}")

    # Run BFS
    bfs_result = bfs_3d_with_metrics(grid, start_node, goal_node)
    if bfs_result:
        bfs_path_length, bfs_time, bfs_nodes, bfs_path = bfs_result
        print(f"BFS - Path Length: {bfs_path_length}, Time: {bfs_time:.4f}s, Nodes
Expanded: {bfs_nodes}")
    else:
        print("BFS - No path found")
        bfs_path = None

    # Run A*
    a_star_result = a_star_3d_with_metrics(grid, start_node, goal_node)
    if a_star_result:
        a_star_path_length, a_star_time, a_star_nodes, a_star_path = a_star_result
        print(f"A* - Path Length: {a_star_path_length}, Time: {a_star_time:.4f}s, Nodes
Expanded: {a_star_nodes}")
    else:
        print("A* - No path found")
        a_star_path = None

    # Visualize the results
    ## visualize_3d_path(grid, start_node, goal_node, path_bfs=bfs_path,
path_astar=a_star_path)
```